

Efficient Random Sampling – Parallel, Vectorized, Cache-Efficient, and Online

Peter Sanders
sanders@kit.edu

Sebastian Lamm
lamm@ira.uka.de

Lorenz Hübschle-Schneider
huebschle@kit.edu

Emanuel Schrade
schrade@kit.edu

Carsten Dachsbacher
dachsbacher@kit.edu

Karlsruhe Institute of Technology, Karlsruhe, Germany

October 18, 2016

Abstract

We consider the problem of sampling n numbers from the range $\{1, \dots, N\}$ without replacement on modern architectures. The main result is a simple divide-and-conquer scheme that makes sequential algorithms more cache efficient and leads to a parallel algorithm running in expected time $\mathcal{O}(n/p + \log p)$ on p processors. The amount of communication between the processors is very small and independent of the sample size. We also discuss modifications needed for load balancing, reservoir sampling, online sampling, sampling with replacement, Bernoulli sampling, and vectorization on SIMD units or GPUs.

1 Introduction

Random sampling is a fundamental ingredient in many algorithms, e.g. for data analysis. With the advent of ever larger data sets (“Big Data”), the number of elements sampled from and even the sample itself can become huge. Often the subsequent processing of the sample is comparatively fast, and thus taking the sample can become a performance bottleneck. Moreover, the speed of a single processor is stagnating so that parallel algorithms are required for efficient sampling. Furthermore, we can observe that only *local* processing yields fast parallel algorithms and promises to scale linearly with the number of processors p . Processor coordination over global memory or even communication over the network quickly becomes a bottleneck [Sanders et al., 2013]. This is particularly true for big data problems which often run on cloud resources with limited communication capabilities, or for high performance computing where the largest configurations are limited by the bisection bandwidth of the network.

In this paper we focus on the classical problem of sampling n numbers out of the range $1..N$ without replacement.¹ In Section 2 we discuss building blocks and previous approaches. Section 3 introduces our divide-and-conquer algorithm for sampling without replacement. We discuss a number of generalizations in Section 4 including online sampling in sorted order, load balancing, uneven distribution of the sampled universe, reservoir sampling, using true randomness,

¹We use the notation $a..b$ as shorthand for $\{a, \dots, b\}$.

achieving deterministic results, sampling with replacement, Bernoulli sampling, and vectorization. After providing details of our implementation in Section 5, Section 6 describes experiments which demonstrate the speed and scalability of our algorithm on both CPUs and GPUs. Section 7 summarizes the results and discusses some applications.

2 Preliminaries and Related Work

Our goal is to efficiently take a sample of size n from the range $1..N$ using p processors. The restriction to the range $1..N$ is without loss of generality: if we want to sample from a general set M of size N , we can view M as an array of size N and the generated numbers will be the indices of the sampled elements. To avoid special case discussions, we will henceforth assume that $n \leq N/2$. For the unusual case $n > N/2$, one can simply generate the $N - n < N/2$ elements that are *not* in the sample. When considering parallel algorithms, we use p to denote the number of processing elements (PEs), which we assume to be connected by a network. PEs are numbered $1..p$.

Algorithm S [Fan et al., 1962, Knuth, 1981] scans the range $1..N$ and generates a uniformly random deviate for each element to decide whether it is sampled. For $N \gg n$ this is prohibitively slow and we are surprised that the algorithm still seems to be widely used, for example by the GNU Scientific Library, GSL (function `gsl_ran_choose`, <https://www.gnu.org/software/gsl/>, version 2.2.1).

Algorithm H is a simple and efficient folklore algorithm that is very good for small n (see also [Ahrens and Dieter, 1985]). The sample is kept in a hash table T which is initially empty. To produce the next sample element, it generates uniform deviates X from $1..N$. If $X \in T$, it rejects X , otherwise X is inserted into T . This algorithm runs in expected time $\mathcal{O}(n)$. Note that T contains random numbers and hence we can use a very simple hash function, such as extracting the most significant $\log n + \mathcal{O}(1)$ bits from the key.² For $n \ll N$ the number of random deviates required is close to n , and we only need uniform deviates. This makes Algorithm H very fast for small n . For large n , however, most hash table accesses cause cache faults, slowing it down considerably.

Algorithm H can also be parallelized. However, the hash table accesses then become global interactions between the processors. The resulting overheads are even larger than the cache faults in the sequential algorithm and cause a severe bottleneck in distributed settings. One also has to be very careful if the resulting algorithm is supposed to be *deterministic*, i.e., the generated sample should be the same in repeated runs with the same seeds for the random number generators: race conditions in remote memory accesses or message delivery can easily lead to differences in the generated sample.

Algorithm D Vitter [Vitter, 1984] proposed an elegant sequential algorithm that generates the samples in sorted order without any need for auxiliary data structures. For generating the next sample, Algorithm D essentially generates an appropriate random deviate that specifies the number of positions to skip. Note that the random deviate changes in each step; using sophisticated techniques based on the rejection method, this can be done in constant expected time.

Algorithm B Ahrens and Dieter [Ahrens and Dieter, 1985] use the observation that taking a Bernoulli sample where each element of $1..N$ is sampled with probability $\rho \approx n/N$ yields a sample

²In this paper $\log x$ stands for $\log_2 x$.

with $n' \approx n$ elements. If this sample is too big it can be repaired by removing $n' - n$ of the elements randomly. By choosing ρ somewhat larger than n/N , one can make the case $n' < n$ highly unlikely and simply restart the sampling process if it does occur. Bernoulli sampling can be implemented efficiently by generating geometrically distributed random deviates to determine how many elements to skip in each step. Algorithm B is faster than Algorithm D because generating geometric random deviates needs less arithmetic operations per element. In Section 4.8 we point out that it may be even more important that the parameters of the generated distribution remain the same, as this makes vectorization possible. A notable difference of Algorithm B to the aforementioned approaches is that elements are not generated online, i.e., we have an initial delay of $\Theta(n)$ before the first sample is generated.

3 Divide-and-Conquer Sampling

We now present our divide-and-conquer approach, beginning with a simple sequential setting before proceeding to its parallel adaptations.

3.1 Sequential Divide-and-Conquer Sampling (Algorithm R)

Our central observation is that for any splitting position ℓ , the number of samples L from the left half $1..\ell$ is distributed hypergeometrically with parameters n (# of experiments), ℓ (# of success states) and N (universe size). Consequently, the number of samples from the right part $\ell + 1..N$ is $n - L$. Algorithm R in Figure 1 gives pseudocode for a sequential divide-and-conquer algorithm based on this idea. The tuning parameter n_0 decides when to switch to the base case. When using Algorithm H, n_0 should be small enough so that the hash table fits into cache. Note that the resulting recursion tree has a size of at most $2n/n_0$. Hence the overall expected running time is $\mathcal{O}(n)$, provided that we use a constant time algorithm for generating hypergeometric random deviates (e.g. [Stadlober, 1990]) and a linear expected time algorithm for the base case.

```

Function sampleR( $n, N$ )
  if  $n < n_0$  then return sampleBase( $n, N$ )           // e.g. using algorithms H or D
   $x := \text{hyperGeometricDeviate}(n, \lfloor N/2 \rfloor, N)$ 
   $A := \text{sampleR}(x, \lfloor N/2 \rfloor)$ 
   $B := \text{sampleR}(n - x, N - \lfloor N/2 \rfloor)$ 
  return  $A \cup \{x + \lfloor N/2 \rfloor : x \in B\}$ 

```

Figure 1: Algorithm R for (sequential) divide-and-conquer sampling without replacement.

3.2 Parallel Divide-and-Conquer Sampling (Algorithm P)

For parallel sampling, we partition the range $1..N$ into p pieces. Let N_i denote the last element in the range associated with processor i , i.e., processor i generates the sample elements that lie in the range $N_{i-1} + 1..N_i$ with $N_0 := 0$. The underlying idea of the parallelization is to adapt Algorithm R in a way such that $\lceil \log p \rceil$ levels of recursion split the original range $1..N$ into the subranges of each

```

Function sampleP( $n', j..k, i, h$ )
  if  $k - j = 1$  then
    use  $h(i)$  to seed the local pseudorandom number generator
     $M := \text{sampleLocally}(n', N_i - N_{i-1} + 1)$  // e.g. using algorithms H, D, or R
    return  $\{N_{i-1} + x : x \in M\}$ 
   $m := \lfloor \frac{j+k}{2} \rfloor$  // middle processor number
   $x := \text{hyperGeometricDeviate}(n', N_m - N_j + 1, N_k - N_j + 1, j..k, h)$ 
  if  $i \leq m$  then return sampleP( $x, j..m, i, h$ )
  else return sampleP( $n' - x, m + 1..k, i, h$ )

```

Figure 2: Algorithm P for sampling n' elements on processors $j..k$ where $i \in j..k$ is the PE executing the function. The initial call on processor i is *sampleP*($n, 1..p, i, h$).

processor. Each processor will follow only a single recursive call – the one whose range contains its local subrange.

Locally, each PE can use any sequential algorithm, however, we have to be careful: on the one hand, processors following the same path in this recursion tree have to generate the *same* random deviates to get a consistent result. On the other hand, random deviates generated in two *different* subtrees have to be independent. With true randomness (e.g. generated using a hardware random number generator [Intel, 2012]) this would require communication to distribute the right random values to the processors (see also Section 4.5). However, using pseudorandomness (as most applications do) allows us to achieve the desired effect without any communication. The idea is to use a (high quality) hash function h as source of pseudorandomness for generating hypergeometric deviates. In the subproblem for PEs $j..k$, the t -th random deviate is $h((j, k, t))$. Figure 2 gives pseudocode where the function *hyperGeometricDeviate* is passed both h and $j..k$ in order to be able to use the technique described above. Within function *sampleLocally*, we can still use an ordinary generator of pseudorandomness which may have a better trade-off between speed and quality than hashing. In order to break the symmetry between the processors, we can seed it with $h(i)$ on processor i .

Another issue is that the processors need access to the global sample numbers N_j . If the universe is evenly distributed between processors (except for the last one if p does not divide n) this is easy. We simply have $N_j = j \lceil n/p \rceil$ for $j < p$ and $N_p = N$. Refer to Section 4.3 for the case of uneven distribution of the universe.

We obtain the following running time for Algorithm P.

Theorem 1 *If $\max_i (N_i - N_{i-1}) = \mathcal{O}(N/p)$ then Algorithm P runs in time $\mathcal{O}(n/p + \log p)$ with high probability.*³

Proof. The proof is easy when we only calculate with expectations. Each PE generates $\leq \lceil \log p \rceil$ hypergeometric random deviates and $\mathcal{O}(n/p)$ samples in expectation.

A bit more care is needed to rule out that rare cases slow down computation on some “unlucky” processor which would then lead to a large overall execution time. Three issues have to be considered:

³I.e., with probability at least $1 - p^{-c}$ for any constant c .

deviations in the number of samples per processor, deviations in the time needed to generate the random deviates, and running time fluctuations within function *sampleLocally*.

The number of samples generated by one processor has a hypergeometric distribution. We exploit that this distribution spreads the elements more evenly than a binomial distribution [Sanders, 1996, Theorem 3.3] and analyze the simpler situation when each sample is independently assigned to processor i with probability $(N_i - N_{i-1} + 1)/N = \mathcal{O}(1/p)$. We thus have a classical balls-into-bin situation that can be analyzed using Chernoff bounds [Hagerup and Rüb, 1990]. These bounds yield exactly what we need – $\mathcal{O}(\log p)$ samples with high probability when $n = \mathcal{O}(p \log p)$ and $\mathcal{O}(n/p)$ samples with high probability when $n = \Omega(p \log p)$.

Fast algorithms for generating hypergeometric deviates [Stadlober, 1990] are often based on a rejection method, i.e., they generate a constant number of uniform deviates, perform a constant amount of computation, and then perform a test that succeeds with constant probability. If the test fails, an independent new trial is performed. Hence, the running time of the generator can be bounded by a constant times a geometrically distributed random variable. However, it is easy to show that the sum of $\mathcal{O}(\log p)$ such random variables is $\mathcal{O}(\log p)$ with high probability.

When using Algorithm D for generating local samples, we can use a similar argument as above – the running time for generating each sample is bounded by a geometrically distributed random variable so that large deviations from the expectation are unlikely. When using Algorithm H, the details of the analysis depend on the tails of the running time distribution of the hash table, but we will get the required short tails of the running time distribution if we allocate enough space – $\mathcal{O}(n/p + \log p)$ – for this table. When using Algorithm R, additional hypergeometric random deviates are generated, but the argument with the geometrically distributed running time again holds. ■

4 Generalizations

4.1 Generating Output in Sorted Order and Online

Note that Algorithm R can easily output the elements in sorted order provided that the base case algorithm generates the samples in sorted order. This is certainly the case when using Algorithm D, and we can also adapt Algorithm H for this purpose. For example, we can maintain the invariant that the samples in the hash table are sorted. This is possible since we use the most significant bits to address the table. We only have to ensure that colliding elements are also sorted. Rather than appending an inserted element k to the end of a cluster of colliding elements, we skip elements smaller than k and then shift the cluster elements larger than k one position to the right. This makes handling clusters of colliding elements somewhat slower, but the overall overhead is small since the clusters are small.

Alternatively, we can insert into the hash table normally, ignoring the keys' order, and sort the hash table afterwards. Since the sorting order is the same as the hash function value, the only thing we have to do is to scan the hash table and sort clusters of colliding elements. This leads to a linear time algorithm since the clusters are small.

It is also easy to modify Algorithm R to generate samples online with constant expected delay between generated samples. We can modify the divide-and-conquer step to split off a range of size $\lceil N \cdot n_0/n \rceil$. Using this splitting in an iterative fashion, we generate samples in batches of expected size of approximately n_0 . This takes time $\mathcal{O}(n_0)$ per batch, i.e., constant time if n_0 is a constant.

The same techniques can be used in a parallel setting. Then each processor generates the elements of its designated subrange of $1..N$ in sorted order.

4.2 Load Balancing

Algorithm R implicitly assumes that all processors are equally fast. However, for various reasons, this may not be the case. For example, we might work with heterogeneous cloud resources, there might be other jobs (or operating system services) slowing down some processors, or uneven cooling might imply different clock frequencies for different processors. In these cases, the slowest processor would slow down the overall computation. This problem can be solved with standard load balancing techniques. We split $1..N$ into $p' \gg p$ jobs (subranges) and use a load balancing algorithm to dynamically assign jobs to processors.

The most widely used load balancing method for such problems uses a centralized master processor to assign jobs to processors. Unfortunately, this increases the running time from $\mathcal{O}(n/p + \log p)$ (Theorem 1) to $\mathcal{O}(n/p + p)$. A more scalable approach is *work stealing* [Finkel and Manber, 1987, Blumofe and Leiserson, 1999]. To employ this approach, we instantiate the concept of a *tree shaped computation* [Sanders, 2002]: We conceptually split the work into very fine grained *atomic* jobs corresponding to ranges of sample values that are expected to contain a constant number of samples (say, n_0). However, initially these jobs are coalesced into p (meta) jobs of about equal size. Now each processor sequentially works on its meta job, one atomic job at a time. Idle processors ask random other processors to split their range of unfinished atomic jobs in half, delivering one half to the idle processor. Note that both splitting off the next atomic job and splitting the remaining range of atomic jobs in half can be done in constant expected time using the division strategy from Algorithm P. The generic analysis in [Sanders, 2002] then yields the same asymptotic running time as in Theorem 1.

4.3 Uneven Distribution of the Sampled Universe

When we sample from a set of elements distributed over processors connected by a network, we may not want to load balance. Rather, we want to use the *owner computes* paradigm – each processor computes those samples that stem from its local subset of elements.⁴ In this situation, each processor i initially only knows its local number of elements L_i .

We address this situation by arranging the processors into a binomial tree [Sullivan and Bashkow, 1977]. Let the processors be numbered $0..p-1$ now. If the binary representation of the processor number i ($\lceil \log p \rceil$ bits) contains k trailing zeroes, it is connected with processors $i + 2^j$ for $j \in 0..k-1$ if $i + 2^j < p$. The connections for each value of j form one level of a binary tree (see also Figure 3). At level $j \in 0..\lceil \log p \rceil$ we get (maximal) subtrees spanning processors $2^j a.. \min(2^j a + 2^j - 1, p-1)$ for $a = 0..p/2^j - 1$. In an upward pass, iterating from $j = 0$ upwards, we compute the sum of the L -values in each of these subtrees. For an inner node let L_ℓ and L_r denote the partial sums for its left and right subtrees, respectively.

Now the number of samples in each subtree is computed in a top down fashion. The root knows that it has to generate $n' = n$ samples. Other nodes receive their n' value from above. An inner node uses a hypergeometric distribution with parameters n , L_ℓ , and $L_\ell + L_r$ to split its n' samples into $n' = n_\ell + n_r$. Then n_ℓ is used for the next smaller subtree locally, while n_r is passed to the

⁴In a hybrid setting, where several shared memory machines are connected by a network, we could still apply load balancing on each shared memory machine.

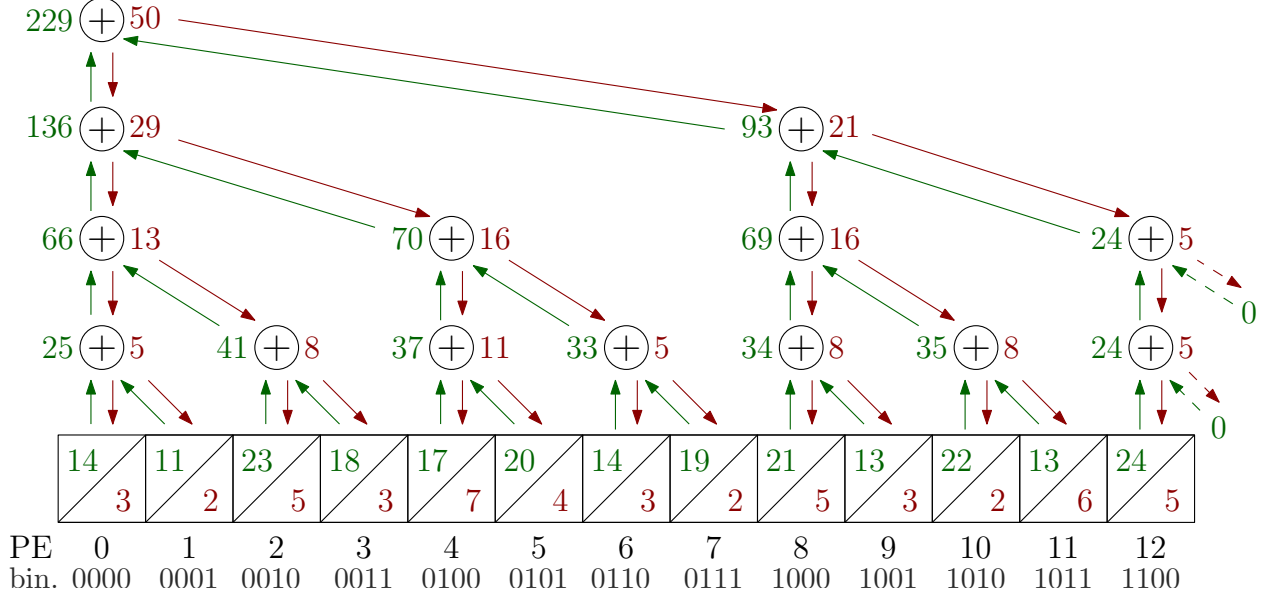


Figure 3: Assigning 50 samples to 13 PEs with a grand total of 229 elements. Element counts (L -values, in green, on the nodes' left side) are added bottom-up from leaves to root, sample counts (n' , in red, right side) are assigned in the opposite direction.

right child as the number of samples to be generated there. To generate independent random values everywhere, the subtree representing processors $a..b$ can use this range as an input for the hash function h from Algorithm P.

4.4 Reservoir Sampling

Reservoir sampling [Vitter, 1985, attributed to Waterman] is a useful technique for maintaining a sample of size n over a data stream. Here, we generalize the single-stream version to multiple distributed streams. Elements arrive at each PE independently. The PEs run classical reservoir sampling locally, using reservoir size n for now. Additionally, each PE keeps track of L_i , the number of elements seen locally. Let N denote the sum of the L_i . Both N and the L_i are functions of time, changing as more data arrives; N has to be computed as PEs only observe their local streams. To draw a global sample, we use the technique of Section 4.3 to determine the number of samples at each PE. Each PE then draws its assigned number of samples from its local reservoir.

We can reduce the local reservoir size if a constant-factor approximation of N is available and we are willing to accept a (very) small probability of failure, δ . A 2-approximation of N can easily be maintained with communication per PE logarithmic in N , p , and $1/\delta$ by randomizing updates. By bounding the tail of the hypergeometric distribution, we obtain that a local reservoir size of $(L_i/N + t) \cdot n$, with $t \geq \sqrt{\ln(p/\delta)/(2n)}$, is sufficient. Should the algorithm fail nonetheless, requesting more samples from a PE than it can provide, we can restart the procedure from Section 4.3, skewing the result slightly. We can ensure that this does not happen in practice by choosing a small enough value for δ . As an example, assuming $L_i = N/p$, $n = 10^5$, $p = 256$, and an extremely conservative $\delta = 10^{-20}$, we obtain $t \approx 0.016$ and a local reservoir size of 2000 elements – fifty times less than required for the naive approach.

To prevent the local reservoirs from changing while the query is running, potentially skewing the result, changes to the local reservoirs are delayed for the duration of the query, e.g. by recording them in a buffer. Once the query is complete, we apply the pending changes. We argue that this is not a performance problem, as queries are fast and the expected number of changes to the reservoir during query time is low.

4.5 Using True Randomness

Now let us assume that each processor has access to some independent physical source of truly random values. In this case, we can use the algorithm from Section 4.3 since it makes every random decision only once and explicitly passes the resulting information to other processors.

4.6 Deterministic Results

For fixed p and h , Algorithm P deterministically and reproducibly generates the same sample every time, which is important to make software using the algorithm predictable, reliable, and testable. If we even want the result to be independent of p , we can use the load balancing method from Section 4.2. In this case, we generate $p' \gg p$ jobs regardless of the actual number of PEs used and then assign the jobs to the PEs (possibly even statically, $\lceil p'/p \rceil$ consecutive jobs for each PE).

4.7 Sampling with Replacement in Various Spaces

Algorithms R and P are easy to adapt to sampling with replacement. The only thing that changes is that the hypergeometric distribution for the divide-and-conquer step has to be replaced by a binomial distribution. Note that this is not restricted to sampling from the one-dimensional discrete range $1..N$, we can also uniformly sample from continuous or higher-dimensional sets as long as we can bipartition the space. For example, for generating random points in a rectangle we can subsequently bisect this rectangle into smaller and smaller rectangles up to some base case. In order to match the size of these base objects to the number of processors, it might be useful to generate $K \gg p$ base objects and to use some kind of load balancing to map base objects to processors. This works similar to the load balancing methods from Section 4.2.

4.8 Relation to Bernoulli Sampling

We want to point out that Bernoulli sampling and sampling without replacement are almost equivalent in the sense that they can emulate each other efficiently. On the one hand, Bernoulli sampling with success probability ρ can be implemented by sampling without replacement if we can first determine how many elements n are sampled by Bernoulli sampling. This number follows a binomial distribution with parameters N and ρ . Then we can use sampling without replacement to choose the actual elements. On machines with slow floating point arithmetics, e.g. microcontrollers, this approach might be faster than generating skip values from a geometric distribution, which requires evaluating logarithms.

On the other hand, Algorithm B [Ahrens and Dieter, 1985] generates n samples without replacement by “repairing” a Bernoulli sample. For this paper, it is important that Bernoulli sampling can also be parallelized in several ways. We can independently apply Bernoulli sampling to subranges of $1..N$. This is the method of choice for distributed memory machines since it requires no communication. On a shared memory machine, we can also generate an array of $(1 + o(1))\rho N$

independent, geometrically distributed random deviates and compute their prefix sums. The values up to N denote the sample. A practically important observation is that the operation needed for this approach has no conditional branches or random memory accesses, and hence can be implemented on SIMD (single instruction multiple data) units of modern CPUs or on GPUs.

To parallelize Algorithm B, we can use it as base case of Algorithm P. We can also use parallel Bernoulli sampling and then use Algorithm P in the repair step.

5 Implementation Details

We have implemented algorithms D, H, R, P, and B using C++.⁵ We use Spooky Hash⁶ as a hash function which generates seeds for initializing the Mersenne twister [Matsumoto and Nishimura, 1998] pseudorandom number generator for uniform deviates.

Algorithm D has been translated literally from the description in [Vitter, 1984].

Algorithm H uses hashing with linear probing [Knuth, 1998] using a power of two as table size. We use two variants for obtaining the entries of the table and for emptying it. The default is to record the positions of inserted elements on a stack. This way, we can retrieve and reset the table elements without having to consider empty entries. In turn, this allows us to make the table size m significantly larger than the final number of entries n in order to speed up table accesses. This does not work when we want to output table entries in sorted order. Here we omit the stack and explicitly scan the table at the end. Furthermore, we allocate n additional table entries to the right so that it becomes unnecessary to wrap around when an insertion probes beyond the m -th table entry. Otherwise, wrapping around could destroy the globally sorted order between clusters (see Section 4.1).

Algorithm R uses Algorithm H as the base case sampler (*sampleBase* in the pseudocode of Figure 1). We do this because Algorithm H is faster than Algorithm D for small subproblems where the hash table fits into cache. This will always be the case if n_0 is chosen appropriately (we use $n_0 = 2^9$ and $m = 2^{12}$). To generate hypergeometric random deviates, we use the `stocc` library⁷, which uses a Mersenne twister internally.

Algorithm P on Blue Gene/Q is parallelized using MPICH 1.5 on gcc 4.9.3. It uses Algorithm R with parameters $n_0 = 2^8$ and $m = 2^{11}$ as local sampling algorithm.

Algorithm B uses Algorithm R for selecting samples to be removed in the repair step. Geometric random deviates are generated using the C++ standard library (`std::geometric_distribution` and `std::mt19937_64`).

We implemented two further variants of Algorithm B. One targets SIMD parallelism within a single CPU-core. The other uses NVIDIA GPUs. The CPU-SIMD version performs best when restricting arithmetics to 32 bits. Therefore we use a smaller maximal universe size of $N = 2^{30}$ there. This version uses the Intel Math Kernel Library MKL v11.3 [Intel, 2015] to generate geometric deviates. Prefix sums are computed by a manually tuned routine using SSE2 instructions through compiler intrinsics.

⁵<https://github.com/sebalamm/DistributedSampling> and <https://github.com/lorenzhs/sampling>

⁶<http://www.burtleburtle.net/bob/hash/spooky.html>, version 2

⁷<http://www.agner.org/random/>, version 2014-Jun-14

The GPU version uses CUDA 7.5, the cuRAND library⁸ for generating geometric random deviates and the Thrust library⁹ for computing prefix sums. Thus, most of the work can actually be delegated to libraries tuned by the vendor. Unfortunately, the repair step, albeit requiring only sublinear work, is difficult to do on the GPU. Therefore it is partially delegated to the CPU. There are various ways to accomplish this but the key point is to do it in a way such that the sample does not need to be transferred to the CPU. Our solution first uses a parallel GPU pass over the sample to count the number n' of prefix sum values $< N$ (see Section 2). Only the single value n' needs to be transferred to the CPU. The CPU then uses Algorithm R to generate $n' - n$ samples from the range $0..n' - 1$. These samples are transferred to the GPU which marks the appropriate positions in the sample array for removal. Finally, the sample array is compacted using the Thrust function `copy_if`.

6 Experiments

Figure 4 compares the performance of the sequential Algorithms D, H, R, and B. These experiments were conducted on a single core of a dual-socket Intel Xeon E5-2670 v3 system with 128 GiB of DDR4-2133 memory, running Ubuntu 14.04. The code was compiled with GNU `g++` in version 6.2 using optimization level `fast` and `-march=native`. We report results for universe size $N = 2^{50}$ and varying n . The number of repetitions was $2^{30}/n$ to achieve equal work for every n . We see that Algorithm H is very fast for small n , but its performance degrades as n grows and the hash table exceeds the cache size. Our new Algorithm R is similarly fast for small n , but the time per sample remains constant as n grows. Thus, it is up to 5 times faster than Algorithm H for very large n . The performance of Algorithm D is also independent of n , but worse than Algorithm R by a factor of 7. A variant of Algorithm R (SR) that generates samples online and in sorted order is still 3.4 times faster than Algorithm D. The portable implementation of Algorithm B (labeled *B* in Figure 4) is faster than Algorithm D but cannot compete with Algorithm R.

This picture changes when looking at tuned architecture specific implementations of Algorithm B. The CPU version (label B_{MKL}) is up to 6 times faster than Algorithm R for large n . For very large n , the GPU version, B_{GPU} , running on an NVIDIA GeForce GTX 980 Ti graphic card, is yet 4.5 times faster. However, it should be noted that a single core of a Xeon E5-2670 v3 uses much less power than an entire GTX 980 Ti – the entire Xeon processor with 12 cores uses about half the power of the graphics card.

Our experiments clearly confirm our expectation that offloading sampling to the GPU for further processing on the CPU is not worthwhile, as the time for transferring the samples from GPU to CPU memory (not pictured in Figure 4) dwarfs the time to take the sample – including transfer, a single core of a modern CPU can generate the samples equally fast. However, it also shows that fast sampling is possible for GPU applications, i.e. if the samples are required on the GPU for further processing.

It is also worth looking at the individual components of the running time of the GPU implementation (we consider the case of $n = 2^{27}$ as an example). The CPU portion and data transfer account for 13.3 % of the total running time, while 86.7 % are spent on computation on the GPU. This time, in turn, is split up as follows. Generating geometrically distributed random numbers using cuRAND takes 25.0 % of the computation time, and calculating a prefix sum over the elements using the

⁸<http://docs.nvidia.com/cuda/curand/>, v7.5

⁹<https://developer.nvidia.com/thrust>, v1.7.0

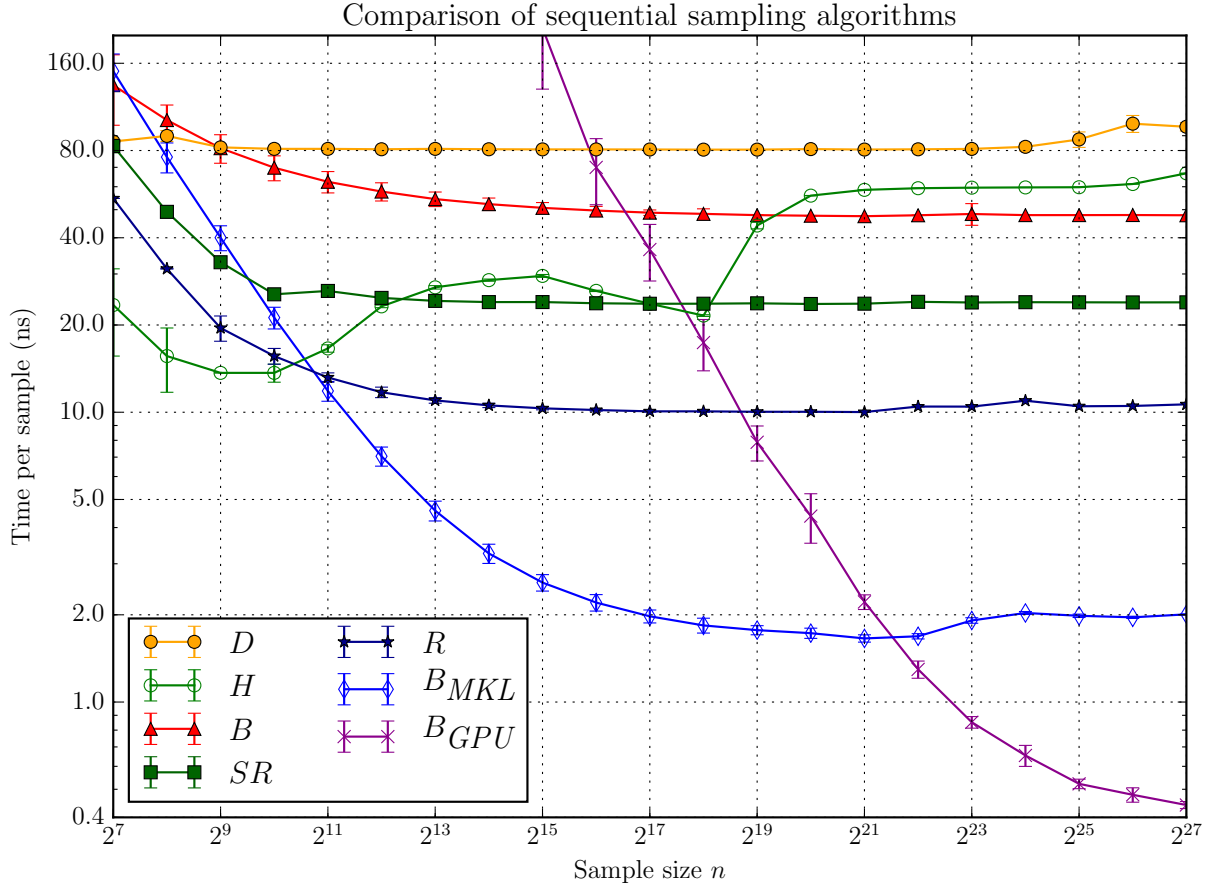


Figure 4: Running time per sample for the sequential algorithms H, D, R, and B. The bars show the standard deviation. The number of repetitions for each algorithm is $2^{30}/n$. For Algorithm R, we use $n_0 = 2^{10}$. SR is Algorithm R with sorted output. B_{MKL} and B_{GPU} are non-portable vectorized implementations of Algorithm B for CPUs using Intel’s Math Kernel Library (MKL) and NVIDIA GPUs using CUDA, respectively.

Thrust function `inclusive_scan` takes another 36.7%. Counting the number of elements $< N$ with `count_if` takes 6.9%. While the time for marking the elements selected by the CPU is negligible at 0.2%, the following compaction with the Thrust function `copy_if` takes another 31.0%.

Algorithm P Figure 5 shows a so-called weak scaling experiment on JUQUEEN, a distributed memory machine. It shows the running time of Algorithm P when keeping local input size n/p constant, measured for different values of this ratio. JUQUEEN is an IBM Blue Gene/Q machine, demonstrating the portability of our code. We used the maximum number of 16 cores per node for these experiments. Performance per core is an order of magnitude lower than on the Intel CPU used for our sequential experiments. A factor of four is more typical for other applications considering the lower clock frequency, older technology, and lower number of transistors used. The remaining factor of 2–3 is mostly due to the fact that our random number generator, a SIMD-oriented Mersenne

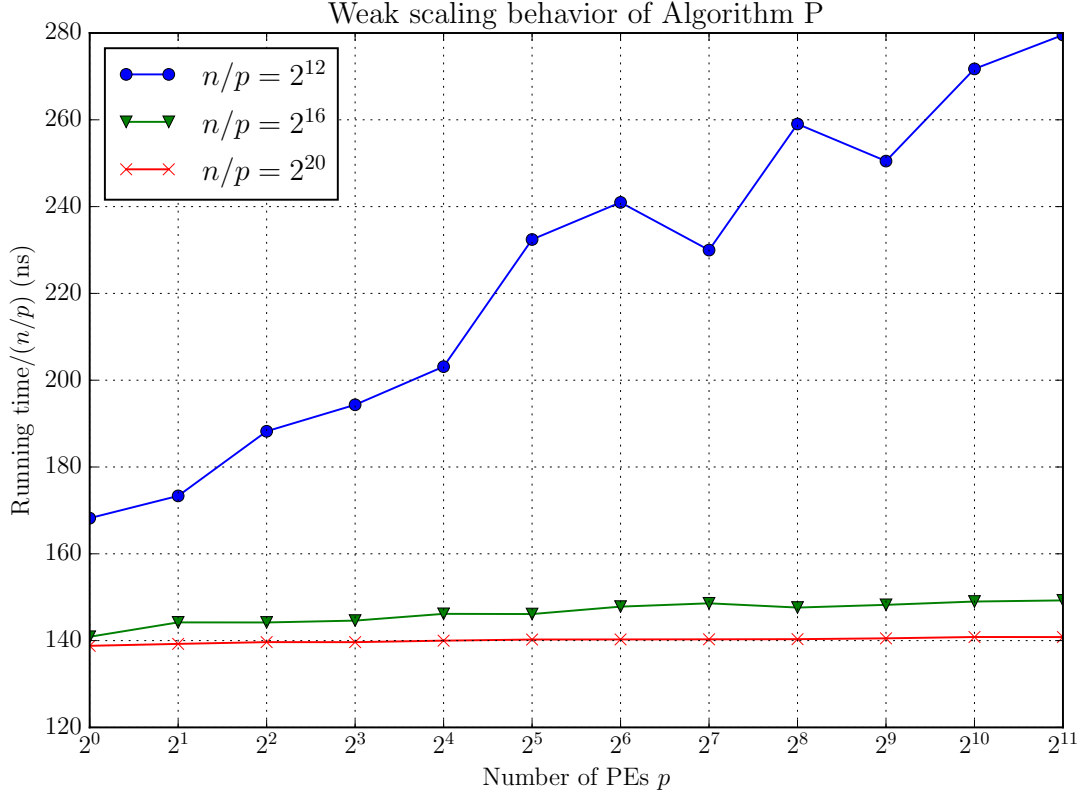


Figure 5: Running time for generating n samples on p processors for different values of n/p using Algorithm P with Algorithm R as local sampler, using $n_0 = 2^8$. The number of repetitions for each value of n/p is $2^{30} \cdot p/n$.

Twister, contains optimizations to make use of the SSE2 units of Intel CPUs. However, it does not have similar optimizations for the QPX instructions of Blue Gene/Q, thus reverting to scalar code. This is compounded further by the lack of autovectorization for Blue Gene/Q in gcc.

On the positive side, we see that the code scales almost perfectly for sufficiently large values of n/p . For the smallest tested value of n/p , 4096, we see a linear increase in running time with an exponential increase in p . This is consistent with the asymptotic running time of $n/p + \log p$.

7 Conclusions

We find it surprising that the seemingly trivial problem of random sampling requires such a diverse set of algorithmic techniques. Moreover, the features of modern computer architectures entail that no single approach is universally best. When n is very small, Algorithm H is both simple and efficient, but for larger n it becomes cache-inefficient. This problem can be overcome by using it for the base case of Algorithm R. A slight generalization of Algorithm R allows for parallelization (Algorithm P). Since this requires no or almost no communication, it is suitable for many parallel models of computation, such as shared memory, distributed memory, or cloud computing. Only some details like load balancing and adaptation to nonuniform data distribution require communication.

On the other hand, we see no reason for using Algorithm S anymore. Algorithm S is fast (only) if N/n is a small constant, but we doubt that it can ever outperform Algorithm R, which needs at most half the number of uniform deviates. In particular, for small N/n we could use a variant of Algorithm H for the base case that uses the key directly to index the table of sampled elements. This avoids the need for handling collisions between samples.

The main point in favor of Algorithm D is that it generates the samples in sorted order and works in an online fashion, i.e., the expected time between generating samples is constant. With the iterative version of Algorithm R described in Section 4.1 we can achieve the same effect, but with a more flexible trade-off between maximum latency between samples and the average cost per sample. From that perspective, our divide-and-conquer technique is a generalization of Algorithm D that allows faster processing and parallelization. Actual real time guarantees of deterministic constant time between subsequent samples seem to be an open problem and neither Algorithm R nor D can offer such guarantees.

Algorithm B is useful because it allows vectorization. Hence, on architectures with fast arithmetics, a tuned version of Algorithm B can outperform Algorithm R. However, the price to pay for that is reduced portability and that samples cannot be generated in an online fashion – it is only after the repair step that we know which samples survive.

To illustrate the usefulness of fast sampling algorithms, we mention a few applications. Generating a random graph in the $G(n, m)$ and $G(n, p)$ model of Erdős and Rényi [Erdős and Rényi, 1959] is equivalent to sampling from the $n(n - 1)/2$ possible edges. Sampling is performed without replacement for $G(n, m)$ and Bernoulli sampling is used for $G(n, p)$. Sample sort [Bluelloch et al., 1991] is a successful example of a parallel sorting algorithm that splits its input based on a random sample. With Algorithm P this is now possible with very low overhead and without resorting to simplified sampling models, which often complicate the analysis.

Acknowledgments

The authors gratefully acknowledge the Gauss Centre for Supercomputing (GCS) for providing computing time through the John von Neumann Institute for Computing (NIC) on the GCS share of the supercomputer JUQUEEN [Stephan and Docter, 2015] at Jülich Supercomputing Centre (JSC). GCS is the alliance of the three national supercomputing centres HLRS (Universität Stuttgart), JSC (Forschungszentrum Jülich), and LRZ (Bayerische Akademie der Wissenschaften), funded by the German Federal Ministry of Education and Research (BMBF) and the German State Ministries for Research of Baden-Württemberg (MWK), Bayern (StMWFK) and Nordrhein-Westfalen (MIWF).

References

- [Ahrens and Dieter, 1985] Ahrens, J. H. and Dieter, U. (1985). Sequential random sampling. *ACM Trans. Math. Softw.*, 11(2):157–169.
- [Blelloch et al., 1991] Blelloch, G. E., Leiserson, C. E., Maggs, B. M., Plaxton, C. G., Smith, S. J., and Zagha, M. (1991). A comparison of sorting algorithms for the connection machine CM-2. In *3rd Symposium on Parallel Algorithms and Architectures*, pages 3–16.
- [Blumofe and Leiserson, 1999] Blumofe, R. D. and Leiserson, C. E. (1999). Scheduling multi-threaded computations by work stealing. *Journal of the ACM*, 46(5):720–748.
- [Erdős and Rényi, 1959] Erdős, P. and Rényi, A. (1959). On random graphs, i. *Publicationes Mathematicae (Debrecen)*, 6:290–297.
- [Fan et al., 1962] Fan, C. T., Muller, M. E., and Rezucha, I. (1962). Development of sampling plans by using sequential (item by item) selection techniques and digital computers. *Journal of the American Statistical Association*, 57(298):387–402.
- [Finkel and Manber, 1987] Finkel, R. and Manber, U. (1987). DIB – A distributed implementation of backtracking. *ACM Trans. Prog. Lang. and Syst.*, 9(2):235–256.
- [Hagerup and Rüß, 1990] Hagerup, T. and Rüß, C. (1990). A guided tour of chernoff bounds. *Information Processing Letters*, 33:305–308.
- [Intel, 2012] Intel (2012). *Intel Digital Random Number Generator (DRNG): Software Implementation Guide*. Intel. <https://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide>.
- [Intel, 2015] Intel (2015). *Intel Math Kernel Library v11.3*. Intel. <https://software.intel.com/en-us/mkl-reference-manual-for-c>.
- [Knuth, 1981] Knuth, D. E. (1981). *The Art of Computer Programming—Seminumerical Algorithms*, volume 2. Addison Wesley, 2nd edition.
- [Knuth, 1998] Knuth, D. E. (1998). *The Art of Computer Programming—Sorting and Searching*, volume 3. Addison Wesley, 2nd edition.

- [Matsumoto and Nishimura, 1998] Matsumoto, M. and Nishimura, T. (1998). Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *TOMACS: ACM Transactions on Modeling and Computer Simulation*, 8:3–30.
- [Sanders, 1996] Sanders, P. (1996). *Lastverteilungsalgorithmen für parallele Tiefensuche*. Dissertation, Universität Karlsruhe.
- [Sanders, 2002] Sanders, P. (2002). Randomized receiver initiated load balancing algorithms for tree shaped computations. *The Computer Journal*, 45(5):561–573.
- [Sanders et al., 2013] Sanders, P., Schlag, S., and Müller, I. (2013). Communication efficient algorithms for fundamental big data problems. In *IEEE Int. Conf. on Big Data*, pages 15–23.
- [Stadlober, 1990] Stadlober, E. (1990). The ratio of uniforms approach for generating discrete random variates. *Journal of Computational and Applied Mathematics*, 31(1):181–189.
- [Stephan and Docter, 2015] Stephan, M. and Docter, J. (2015). Jülich Supercomputing Centre. JUQUEEN: IBM Blue Gene/Q Supercomputer System at the Jülich Supercomputing Centre. *Journal of large-scale research facilities*, A1.
- [Sullivan and Bashkow, 1977] Sullivan, H. and Bashkow, T. R. (1977). A large scale, homogeneous, fully distributed parallel machine, i. In *Proceedings of the 4th Annual Symposium on Computer Architecture*, ISCA '77, pages 105–117, New York, NY, USA. ACM.
- [Vitter, 1984] Vitter, J. S. (1984). Faster methods for random sampling. *Commun. ACM*, 27(7):703–718.
- [Vitter, 1985] Vitter, J. S. (1985). Random sampling with a reservoir. *ACM TOMS*, 11(1):37–57.